ISSN 2007-9737

Proactive Load Balancing to Reduce Unnecessary Thread Migrations on Chip Multi-Processor (CMP) Systems

Ulises Revilla-Duarte[1], Marco A. Ramírez-Salinas[1,*], Luis A. Villa-Vargas[1], Andrei Tchernykh[2]

[1] Instituto Politécnico Nacional, Centro de Investigación en Computación, Mexico

[2] Centro de Investigación Científica y de Educación Superior de Ensenada, Departamento de Ciencias de la Computación, Mexico

{mars, lvilla}@cic.ipn.mx, chernykh@cicese.mx, urevillaa09@sagitario.cic.ipn.mx

Abstract. For a Linux operating system scheduler that is aware of Chip Multi-Processor (CMP) systems to carry out load balancing is extremely important and quite challenging. The scheduler is a vital component of the Linux kernel responsible for choosing the next thread to run and allocating to a processor core for execution. This process involves primarily a load-balancing procedure that provides the thread migration between the cores of a CMP system. A modern Linux scheduler is designed to obtain the best possible performance while ensuring a fair allocation of the processor cores' time among the normal (non-real-time) threads, which is known as Completely Fair Scheduling (CFS) policy. However, this policy collaterally can cause a relentless execution of the load-balancing procedure, and therefore, an excessive number of thread migrations. According to the literature, an increased cache invalidation, scheduling latency, and power consumption are issues inherent to this. In this paper, we propose and evaluate a proactive load-balancing (PLB) algorithm to reduce unnecessary thread migrations on CMP systems. By comprehensive experimental analysis, we show that our PLB algorithm reduces the number of thread migrations by 43.8% on average without degradation of performance.

Keywords. Linux CFS, load balancing, perf_event tool, PMU counters, chip multi-processor.

1 Introduction

The scheduler is a crucial component of the Linux kernel responsible for choosing the next thread to run and allocating to a processor core for execution [35, 37, 1, 13, 22]. This process involves primarily a load-balancing procedure that provides the thread migration between the cores of a CMP system. For a modern Linux scheduler that is aware of CMP systems to carry out load balancing is extremely important and quite challenging.

"The load-balancing procedure is based on a number of criteria of varying relative importance. The scheduling algorithm policy determines the importance of each of the criteria. Unfortunately, it is impossible to design an algorithm that fits in all the criteria simultaneously; trying to improve performance according to one criterion would adversely affect the expected performance by another" [14].

Nowadays, the Linux scheduling policy is designed to obtain the best possible performance while ensuring a fair allocation of the processor cores' time among the normal (non-real-time) tasks[1]. It is known as Completely Fair Scheduling (CFS) policy [20, 24, 28]. However, this policy

[1]Linux uses the term "task" to refer to both an entire process and a process thread.

collaterally can cause a relentless execution of the load-balancing procedure and an excessive number of thread migrations [21].

It results in several disadvantages, such as an increased cache invalidation, scheduling latency, and power consumption [8, 21]. Let us briefly expose how the Linux scheduler's main functions work to gain a better understanding of the leading role of load balancing in CFS performance. A modern Linux kernel scheduler is composed of two functions: `scheduler_tick()` and `schedule()`.

The `scheduler_tick()` function is used by the kernel's timer system to periodically update the process' runtime statistics as well as to mark processes needing rescheduling (e.g., a higher priority task has just showed up, or a running task has simply spent too much time on a core). It is named the Periodic Scheduler.

The `schedule()` function is called by `scheduler_tick()` after a current process has been marked as needing rescheduling to fairly decide which process most deserves to run next. The current task itself may also call `schedule()` when it has to wait for a resource or an event's non-blocking signal in order to voluntarily yield, in the meantime, its core's time to another task.

A task temporarily yields its core's time without being blocked—the task remains in the TASK_RUNNING state—by calling the `sched_yield()` system call which ends up calling `schedule()` (ergo, Linux is a preemptive multitasking operating system).

The `schedule()` function is named the Main Scheduler. It is aware of CMP systems (a.k.a. homogeneous or symmetric multi-core systems). In the process of choosing the next task to run, `schedule()` carries out load balancing of both real-time (RT) and normal (CFS) tasks.

RT tasks are assigned the highest static priorities in the system (by default range from 0 to 99) in order to receive enough processing time to meet critical time constrains.

`schedule()` calls the function `pull_rt_task()`[2] to pull RT tasks from busier cores and distribute them according to their priorities among a group of RT subqueues (`struct rt_rq`) embedded as a field in the current core's run queue (`struct rq`).

CFS tasks are user tasks (including those of `root`) and kernel daemons that share a processor core according to their dynamic priorities given by nice values (numbers from -20 to 19 with a default of 0). As soon as `schedule()` is called, it disables the kernel preemption making sure not to be interrupted.

A run queue (`struct rq`) is a per-core, linear set of fields holding different types of data and statistics to handle the core's runnable tasks. The run queue is the primary scheduling data structure on which the Linux scheduler operates.

On the per-core run queue a CFS subqueue (`struct cfs_rq`) is built as a red-black tree data structure where tasks are arranged according to their runtime (given by the `vruntime` parameter). Tasks that have not run a relative long time are placed on the lower-left side of the tree. The left-most task is always picked to run next. Also, the run queue holds one RT subqueue (`struct rt_rq`) implemented as a doubly linked list per static priority level (0-99).

To balance CFS tasks, the Main Scheduler first checks the per-core `cfs_rq->nr_running` flag for load imbalance. This flag keeps track of the number of ready-to-run CFS tasks queued in a core's CFS subqueue. Then, `schedule()` calls the function `idle_balance()` which calls the function `load_balance()` to pull CFS tasks from bustling cores and insert them into the CFS subqueue (`struct cfs_rq`) in the current core's run queue (`struct this_rq`).

Once this pull-load balancing is done, the Main Scheduler picks the next task to run and performs the context switch. At this point, the Main Scheduler must be sure that there are no RT tasks in the current run queue waiting to be dispatched.

[2]From kernel versions 2.6.27 to 3.14.79 `pre_schedule_rt()` was used as an enveloping function for `pull_rt_task()`. Recently, from version 3.15.10 to the current stable version 4.16.6, `pull_rt_task()` is included in the `pick_next_task_rt()` function and invoked prior to pick the next rt task to run.

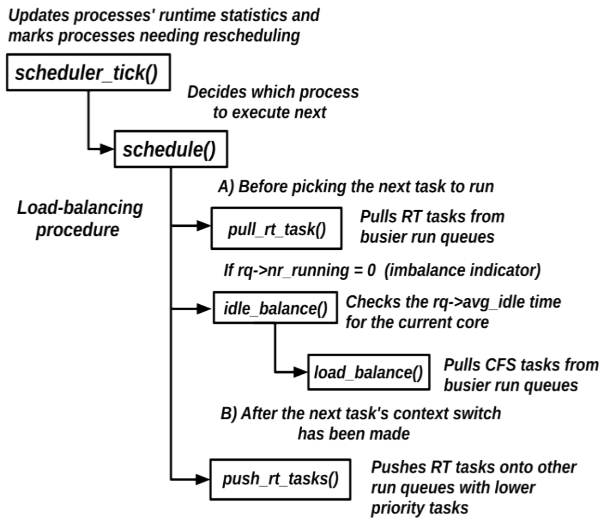
Updates processes' runtime statistics and marks processes needing rescheduling
scheduler_tick()
Decides which process to execute next
schedule()
Load-balancing procedure
A) Before picking the next task to run
pull_rt_task()
Pulls RT tasks from busier run queues
If rq->nr_running = 0 (imbalance indicator)
idle_balance()
Checks the rq->avg_idle time for the current core
load_balance()
Pulls CFS tasks from busier run queues
B) After the next task's context switch has been made
push_rt_tasks()
Pushes RT tasks onto other run queues with lower priority tasks

Fig. 1. Main scheduler's functions involved in load balancing

In order to be executed promptly, the Main Scheduler calls the function `push_rt_tasks()`[3] to push RT tasks, if any, from the current core's run queue onto the run queue of other cores with lower priority tasks (or even experiencing a lack of tasks).

Just after this push-load balancing is done, the Main Scheduler becomes preemtable and checks if the reschedule flag bit (`TIF_NEED_RESCHED`) in the thread information structure (`struct thread-info`) of the current task is set. If set, the search for a new task starts over. If not, `schedule()` exits.

Main Scheduler's functions involved in load balancing are shown in Fig.1. When a processor core is allocated to a new task, the previous task that was running on that core has either gone to the ready or blocked state [31], and can be migrated to another core when a new pull-load balancing operation is performed.

[3]From kernel versions 2.6.27 to 3.13.11 `post_schedule _rt()` was used as an enveloping function for `push_rt_tasks()`. The Main Scheduler now invokes `push_rt_tasks()` through the `balance_callback()` function just after context switching (from version 3.14.79 to the current stable version 4.16.6). `balance_callback()` is part of a novel mechanism which has added a new field for a `callback_head` data structure straight in the core's run queue. `struct callback_head` includes a `void (*func)` field that allows a faster handling of the callback functions `push_rt_tasks()` and `pull_rt_task()`.

Whenever load imbalance is detected, the load-balancing procedure is triggered to distribute the system load among the cores in a homogeneous multi-core processor, which results in excessive task migrations and the consequent drawbacks mentioned earlier.

On the other hand, in accordance with the literature [6, 11, 33, 44], threads' contention for shared resources on a multi-core processor is the major cause of system performance drop. This paper proposes and evaluates a proactive load-balancing (PLB) algorithm for Linux on CMP systems to avoid a decrease in performance due both to contention among the threads for shared resources and excessive thread migrations.

Our PLB algorithm keeps a high level of system performance by proactively averting contending threads from running concurrently as well as by reducing unnecessary thread migrations at runtime. We propose runtime-updated IPC thresholds, which are the basis of the operation of the proactive load balancing. Also, a complementary support algorithm to migrate threads on CMP systems is designed.

The PLB algorithm takes advantage of the Performance Monitoring Unit (PMU) accessible from each core of a modern multi-core processor, and the perf_event tool, a powerful profiling subsystem included in the Linux kernel since version 2.6.31. To meet our proactive load-balancing criterion, our algorithm carries out the following actions at runtime:

1. Configuration of the per-core PMU counters to read different performance-event samples simultaneously at constant time intervals; namely, those corresponding to the Instructions Retired, Unhalted Core Cycles and Thread Migrations events.

   Instructions Retired is the number of fully executed instructions and Unhalted Core Cycles is the number of cycles executed on the core (when the core was not in HALT state), i.e. it shows the total elapsed cycles. These events are sampled for each application thread as a part of the workloads launched separately.

2. Obtaining the Instructions Per Cycle (IPC) statistic which reflects the system performance by using the samples of the Instructions Retired and Unhalted Core Cycles events. In this way, IPC is obtained as follows:

   IPC = instructions retired / unhalted core cycles

   The initial value for the IPC-event (or performance) threshold is set to the first obtained IPC value.

3. Comparison of the subsequent IPC values with the current IPC-event threshold while the workload is running.

4. Reacting proactively to avoid performance ramp-down based on the result of this comparison.

These actions allow to proactively decide whether a running thread must be migrated to another core, and whether its current performance threshold value (previously-sampled IPC value) needs to be updated depending on the result of the comparison.

If a current thread's IPC count is below of its corresponding performance threshold value, the thread is migrated. If this count is higher, then the current performance threshold value is updated to this new IPC count (i.e., runtime-updated IPC thresholds are used).

In this way, only when a runtime-updated IPC threshold is not reached, our algorithm triggers the migration of contending threads in order to find couples of co-running threads that do not contend (or contend as little as possible) for shared resources on the cores, and therefore, leading back to a high level of system performance.

We thereby say that the algorithm obeys a criterion that proactively avoids contending threads from running concurrently.

Threads composing each workload are initially bound to a single core (e.g., core0) as the startup configuration. Our PLB algorithm is implemented at user level, which is sufficient for the accurate assessment [2].

In Section 8, we present a comparative table of different workloads when they run on Linux, first using the original (unmodified) Linux load-balancing procedure, and then merging our algorithm into the Linux kernel. This table shows that the number of thread migrations is significantly reduced (by 43.8% on average) without harming system performance when the PLB algorithm is used.

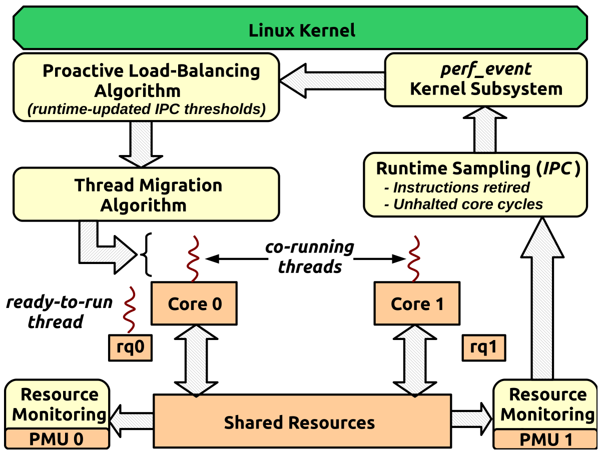

Fig. 2. Architecture of our proactive load balancer

Fig.2 illustrates the architecture of the proposed PLB algorithm. It shows its main components (rounded boxes) interacting with a dual-core CMP system (squared boxes). Next, we briefly describe each component addressed with more detail throughout this article.

– **Resource Monitoring through the PMU**: The PMU (Peformance Monitoring Unit) included within each core is composed of a special set of counting registers (Section 3) that we have configured to count the number of instructions that are fully executed (Instructions Retired), elapsed cycles (Unhalted Core Cycles) and (Thread Migrations).

– **Runtime Sampling (IPC)**: PMU registers are used to collect performance events sampled at regular time intervals at runtime (Section 3).

– **perf_event Kernel Subsystem**: PMU registers setup, runtime performance-event sampling and registers reading are all done through perf_event (Section 3).

– **Proactive Load-Balancing Algorithm**: Our algorithm uses runtime-updated IPC thresholds that indicate the minimum IPC values that must be reach at any sampling instant to hold the system performance at a steady high level.

Table 1. Main features of the example CMP system

Processor	Intel Core^{TM}i5 660 @ 3.33GHz
	Number of Cores 2 physical
	Number of Threads 4 (2 BIOS disabled)
	L1 Cache 2 x 32 KB
	L2 Cache 2 x 256 KB
	L3 Cache 4096 KB
Micro-architecture	Intel Westmere (Codename Clarkdale)
Main Shared Elements	L3 Cache, IM and PCI-e Controllers, and DMI and FDI interfaces
PMUs	Core, uncore and offcore MSRs register sets
Memory Size	4 GB
Memory Type	DIMM DDR3 Syncronous 1066 MHz (2 x 2GB)
Operating System	Ubuntu-GNU/Linux vanilla Linux kernel 3.1.2-SMP x86_32
Compiler	gcc version 4.5.2-8ubuntu4

Therefore, we rely on these threshold values to proactively decide whether to migrate a thread (Section 6).

– **Thread-Migration Algorithm**: If a thread needs to be migrated, our complementary support algorithm designed to migrate threads on a CMP system is invoked (Section 5). Next, the effect of this migration on the system performance is monitored and our PLB algorithm again decides whether to migrate a certain thread in order to maintain high system performance.

Table 1 summarizes the main features of the example CMP system used. The vanilla[4] Linux kernel version 3.1.2 is run on the Intel Corei5 660 processor with codename Clarkdale based on the Intel Westmere microarchitecture [7, 16] whose virtual cores (i.e., hyper-threading) have been disabled from the BIOS, thus having a CMP system with only 2 physical cores. The remainder of this paper is organized as follows: Section 2 briefly surveys related work. Section 3 describes at length the research framework used. Section 4 is devoted to workload selection.

[4]The standard Linux kernel available on the kernel.org web page.

Section 5 unveils the design stages of our thread migration support algorithm. Section 6 explains in detail the implementation of the algorithm that embodies our proposed proactive approach to perform load balancing. Section 7 delineates the evaluation experiments for the PLB algorithm. Section 8 reports the results. Finally, Section 9 concludes the paper and provides an avenue for future work.

2 Related Work

The design of scheduling algorithms that are aided by statistics collected via multi-core architecture-specific performance monitoring counters at run time to avert shared resource contention has been proposed in previous research. For this, they use either the Oprofile [26] or the Perfmon2 [29] external monitoring tools that are no part of the vanilla Linux kernel.

These algorithms aim to minimize the contention for the different shared resources within a CMP processor, such as the L2 or L3 caches, the system bus, the instruction queue, the core itself, and so on, therefore improving the overall system throughput.

Next, we present some previous work that has been done to implement scheduling algorithms that tackle the problem of shared resource contention in today homogeneous multi-core architectures:

Zhang X. et al. [43] developed a flexible framework for Throttling-Enabled Multi-Core Management (TEMM), which efficiently finds an optimal hardware throttling configuration for a user-specified resource management objective.

“It can support a variety of objectives for fairness, quality-of-service, overall performance, and power optimization. Throttling configuration refers to the settings of the platform-specific registers involved with the duty cycle modulation and dynamic voltage and frequency scaling (DVFS) mechanisms, originally designed for power management within processors.

TEMM searches for a reference configuration based on model predictions, and iteratively refines the search with a broad set of previously executed configuration samples. This search stops when a high-quality throttling configuration that meets the objective is found". Sáez J.C. et al. [30] designed a non-work- conserving framework (i.e., a core may be idle at any time) to improve priority enforcement based on statistical information collected through hardware performance monitoring counters (PMU).

"When multiple threads run simultaneously, the system tries to detect changes in the behaviour of high priority (HP) threads that comes from negative interactions with other low-priority (LP) threads. Those changes trigger CPU disabling actions that temporarily block the potentially incompatible LP threads".

Herdrich A. et al. [15] adapted rate-based techniques (clock modulation and frequency scaling) that are employed to address power management and cache/memory Quality of Service (QoS) issues.

The QoS term refers to the ability to guarantee a certain level of performance. Basically, what they do is to regulate the time the core is active and/or its working voltage and frequency (DVFS technique) if it is running a low-priority task that harms the performance of a high-priority task due to system cache or memory contention.

Shi Q. et al. [32] proposed both a load-balancing algorithm based on the construction of scheduling domains by taking shared L2 cache into account and the design of load vectors to weigh the processor core's workload. Their goal is to reduce L2 cache misses (so main memory accesses are also reduced), and therefore, decrease the total execution time of threads.

Lim Q. et al. [21] implemented an operation-zone-based load balancer to improve the performance of multi-core systems at runtime. It provides three multi-core load-balancing policies based on the CPU employment.

"The cold zone policy loosely performs load-balancing operations; it is adequate when the CPU utilization of most tasks is low. The hot zone policy performs load-balancing operations very actively, and it is adequate for high CPU use. The warm zone policy takes the middle between the cold zone and the hot zone".

Our research work proposes a proactive approach to perform load balancing of software threads on homogeneous multi-core processors (i.e., CMP). Our proactive approach is primarily based on runtime-updated IPC thresholds that we devised and used in our decision-making model (Section 6) in order to reduce task migrations originated in the Linux scheduler.

On the example CMP machine used (Table 1), our PLB algorithm maintains two different threads from each workload running concurrently as long as it results in the least shared resource contention, and therefore, to the same extent, thread migration is reduced; thus helping to improve system performance.

Our work relies heavily on the performance monitoring subsystem of the Linux kernel, perf_event, to implement our routines that simultaneously monitor different performance events at runtime—thus providing valuable insight into how to use and configure perf_event.

In a first intance, we developed a complete workload-launcher tool that we used both to synchronously launch workloads made up of various CINT speccpu2000 benchmarks on the example multi-core system and to collect the resulting statistical data from a special set of performance-event counters located within each core's PMU (Performance Monitoring Unit) in the CMP processor.

Our results show that the number of migrations performed on the application threads (benchmarks) that make up the workloads used is significantly reduced (by 43.8% on average) without degradation of performance when our PLB algorithm is utilized.

3 Research Framework

This section describes the research framework used for the implementation of our PLB algorithm. Our research framework consists mainly of both the Performance Monitoring Unit (PMU) included in each core of a multi-core processor and the perf_event profiling tool available in the recent versions of the Linux kernel.

They are used jointly to implement our IPC-based decision-making model as well as to design the procedure for carrying out properly the sampling of different performance events simultaneously at runtime, which are the essential

Table 2. An excerpt from our code to collect instructions, cycles and cpu-migrations events in a single monitoring session

(1)	attr.type = PERF_TYPE_HARDWARE;
(2)	attr.size = sizeof(struct perf_event_attr);
(3)	attr.disable = 1;
(4)	attr.config = PERF_COUNT_HW_INSTRUCTIONS;
(5)	fd = perf_event_open(attr, pid, 0, -1, 0);
(6)	fd1 = perf_event_open(attr, pid, 1, -1, 0);
(7)	attr.config = PERF_COUNT_HW_CYCLES;
(8)	fd2 = perf_event_open(attr, pid, 0, fd, 0);
(9)	fd3 = perf_event_open(attr, pid, 1, fd1, 0);
(10)	attr.type = PERF_TYPE_SOFTWARE;
(11)	attr.config = PERF_COUNT_SW_CPU_MIGRATIONS;
(12)	attr.exclude_kernel = 0;
(13)	fd4 = perf_event_open(attr, pid, 0, fd, 0);
(14)	fd5 = perf_event_open(attr, pid, 0, fd1, 0);

parts of our algorithm. Programming details of the PMU counters and perf_event are also explained in this section. On the other hand, the different major program elements which make up the scheduler such as its main data structures and fuctions have been studied at length directly from the Linux kernel.

Basically, we mostly used the TOMOYO Linux Cross Reference [9], a very helpful web-based tool, to navegate and analyze extensively the vanilla kernel scheduler source code. Also, the Open MPI Portable Hardware Locality tool (`hwloc`) [5, 12] is first utilized to determine our system's topology and object numbering (`lstopo`), and then to bind threads onto processor cores (`hwloc-bin`).

3.1 The Performance Monitoring Unit

Processors supporting Intel 64 and IA-32 architectures have a Performance Monitoring Unit (PMU) consisting of a collection of Performance Monitoring Counter registers (PMCs) and Performance Monitoring Event registers (PMEs) [2, 25, 11, 17, 39]. PMCs and PMEs are implemented as Model Specific Registers (MSRs). They are accessed via the RDMSR and WRMSR instructions.

PMCs are used to collect event counts or serve as hardware buffers, so they are named Counter MSRs. PMEs are used to indicate what events need to be monitored, so they are named Event Programming MSRs. The number of MSR registers that make up the PMU depends on the processor model.

A monitor is defined to be a combination of a PME for the configuration and one or more PMC registers for collecting data. A counting monitor need only one PMC register.

Therefore, the counting monitors can each be programmed to count one event at a time. A monitoring session consists of several steps that must be followed to collect valid measurements. Those steps can be summarized as follows [25]:

i) Program the monitors (paired PME and PMC registers).
ii) Enable the monitors.
iii) Run the code to be monitored.
iv) Disable the monitors.
v) Collect results.

Each core built on a CMP chip has its own register bank which contains the MSR registers that make up the PMU [17]. The PMU and other registers in the register bank are grouped together to form the architectural state of a process thread.

That is, the architectural state is the set of registers within each core in the CMP processor that holds the state of its respective running subprocess. Therefore, on a CMP processor, each running thread has its own independent architectural state.

When a thread migrates from one core to another, its PMU state also moves. That is, counts of different events collected in the MSR registers in the source core's PMU are replicated into the same type of registers in the target core's PMU. It is this important design feature of multi-core processors that allowed us to implement code to follow a thread from one core to another without loosing information on the accounts of events.

Table 3. Workloads made up of CINT speccpu2000 benchmarks

CINT2000 workloads	
W1 (gzip gcc mcf)	**W11** (gzip bzip2 eon)
W2 (bzip2 gcc mcf)	**W12** (bzip2 eon crafty)
W3 (eon gcc mcf)	**W13** (gzip eon crafty)
W4 (crafty gcc mcf)	**W14** (bzip2 gzip crafty)
W5 (gzip gcc bzip2)	**W15** (eon gzip mcf)
W6 (eon gcc bzip2)	**W16** (bzip2 gzip mcf)
W7 (crafty gcc bzip2)	**W17** (crafty gzip mcf)
W8 (gzip gcc eon)	**W18** (eon bzip2 mcf)
W9 (crafty gcc eon)	**W19** (crafty bzip2 mcf)
W10 (gzip gcc crafty)	**W20** (crafty eon mcf)

3.2 The Linux perf_event Kernel Subsystem

Perf_event is a performance monitoring tool merged into the Linux kernel from version 2.6.31 [10, 40, 41]. The principal goal of perf_event is to provide Linux with the support needed to effectively utilize the PMU, thus allowing an advanced performance analysis.

Currently, it is a powerful kernel subsystem increasingly used in the research and development of new computer systems such as multi-core architectures. Support for the latest architectures is added according to new kernel versions.

The perf_event tool includes plenty of commands to collect and analyze performance and trace data. It can measure both hardware and sofware events. Software events are those that originate in the kernel. Some examples are: the number of context-switches, cpu-migrations or page-faults.

Hardware events are micro-architectural events such as the number of elapsed cycles, instructions retired, L1 cache misses, etc. The perf_event interface (API), `perf_event_ open()` (file `/tools/perf/perf.h`), wraps a single system call which supports a set of requests to configure, measure and collect performance monitoring information. It mainly provides a mechanism to read and write PMU registers.

By means of this system call, the PMU registers can be read during the execution of an application. Therefore, event samples can be obtained at runtime. This system call has the following prototype:

```
int perf_event_open(struct perf_event_attr
*attr, pid_t pid, int cpu, int group_fd,
unsigned long flags);
```

A description of its arguments can be found in the perf_event documentation (file `/tools/perf/Documentation`) and the references [10, 40]. The `perf_event_attr` structure is comprised of several attribute fields used to provide detailed configuration information for the event being created. The perf_event interface selects a PMU's counting monitor and configures its PME register based on the event to be monitored (given by the `attr.config` attribute).

It then returns an integer which is the file descriptor (`fd`) of the corresponding PMC register (counter) where the performance event counts will be collected. The `fd` is used to access the PMC register via standard system calls such as `read()` which is used to read the counter or `ioctl()` which is used to perform the counter input/output operations: reset, enable and disable.

Next, we show how the `perf_event_open()` system call is configured in order to implement a single monitoring session that collects hardware events such as `instructions` and `cycles` as well as a software event such as `cpu-migrations` simultaneously for both system cores.

An excerpt from our code to measure these events is shown in Table 2. Line (1) specifies the `type` attribute of the events to be measured which can be hardware or software type. As `instructions` and `cycles` are collected the hardware type is specified.

Line (2) sets the `size` attribute to the `attr` structure size. Line (3) sets the `disable` attribute to its default value, which is 1, to emphasize that the counter must start out disabled (due to synchronization reasons as discussed in Section 6). Then, line (4) introduces the `config` attribute which is nothing else but the name of the event to be measured. This attribute is set to collect the `instructions` event.

Lines (5) and (6) define the system calls used to count the number of instructions retired on both cores (the third argument is `cpu = 0` for core0 and `cpu = 1` for core1). Additionally, the `group_fd` argument has been set to -1 to establish fd and fd1 as the group leaders for core0 and core1 respectively. Group leaders are used to collect different events as a unit for the same set of instructions that are fully executed (i.e., instructions retired). Line (7) shows the corresponding value for the `config` attribute to measure the `cycles` event. Lines (8) and (9) define the system calls used to count the number of elapsed cycles on both cores. Here, the `group_fd` argument is set to `fd` and `fd1`, the file descriptors for the group leaders.

Next, both the `type` and `config` attributes are changed to measure the `cpu-migrations` event. Line (10) now specifies the software type. Line (11) shows the right name for this event. As `cpu-migrations` is an event that happens in kernel space[5] (ergo, also recorded in the `se.nr_migrations` field of the task descriptor), the `exclude_kernel` attribute is changed from its default value of 1 to 0 to include events taking place in kernel space.

Line (12) shows the new value for this attribute. The system calls to measure `cpu-migrations` on both cores also have the `group_fd` argument set to `fd` and `fd1` as shown in lines (13) and (14). Thus, the `instructions`, `cycles` and `cpu-migrations` events are collected as a unit for the same set of instructions retired. Finally, for all events, the `pid` argument is set to the id number of the process thread to be monitored and the `flags` argument is set to zero.

4 Workload Selection

Workloads made up of different combinations of three CINT speccpu2000 benchmarks [36] were previously characterized using the vanilla Linux kernel 2.6.32.10 patched with the Perfmon2 profiling tool [29] on an Intel Core2 Duo E6550 multi-core processor [16].

[5]Linux divides virtual address space into two parts known as kernel space and user space (also called kernel mode and user mode respectively).

Table 3 presents our workloads and Fig.3 shows the bar graphs that result from their characterization using some key metrics. These metrics are:

a) **Instructions Per Cycle (IPC)**: fully executed instructions divided by the total CPU cycles: IPC = instructions retired / unhalted core cycles

b) **L2_MISSES**: data and instruction misses at second level (L2) cache. On the Intel Core2 Duo processor, the L2 cache is a unified cache that is shared by both cores to serve L1 cache misses of instructions and data.

c) **BUS_TRANS_MEM:BOTH_CORES**: Memory Bus Transactions due to both cores. That is, memory requests initiated by any core on the system bus.

d) **INST_QUEUE:FULL**: cycles during which the instruction queue is full. The instruction queue is a unit where instructions wait until they are ready for execution. An instruction is ready for execution when its operands have already been computed.

As can be seen in Fig.3, a smaller number of instructions per cycle is executed for workloads W1 to W4 and W15 to W20. Also, the number of L2 cache misses is too large for such workloads. Furthermore, both the number of memory requests and the number of cycles during which the instruction queue is full are also too large for these same workloads.

This indicates that the benchmarks composing workloads W1 to W4 and W15 to W20 contend with at least one of their co-runners for shared resources intensely. In particular: the L2 cache, the system bus and the instructions queue. Therefore, such workloads are regarded as best suited to carry out experiments in which a stress capacity for our CMP system is required.

5 Design of the Complementary Support Algorithm to Migrate Threads

The kernel uses the `sched_setaffinity()` system call to provide a different mask of cores (`new_mask`) to a task.

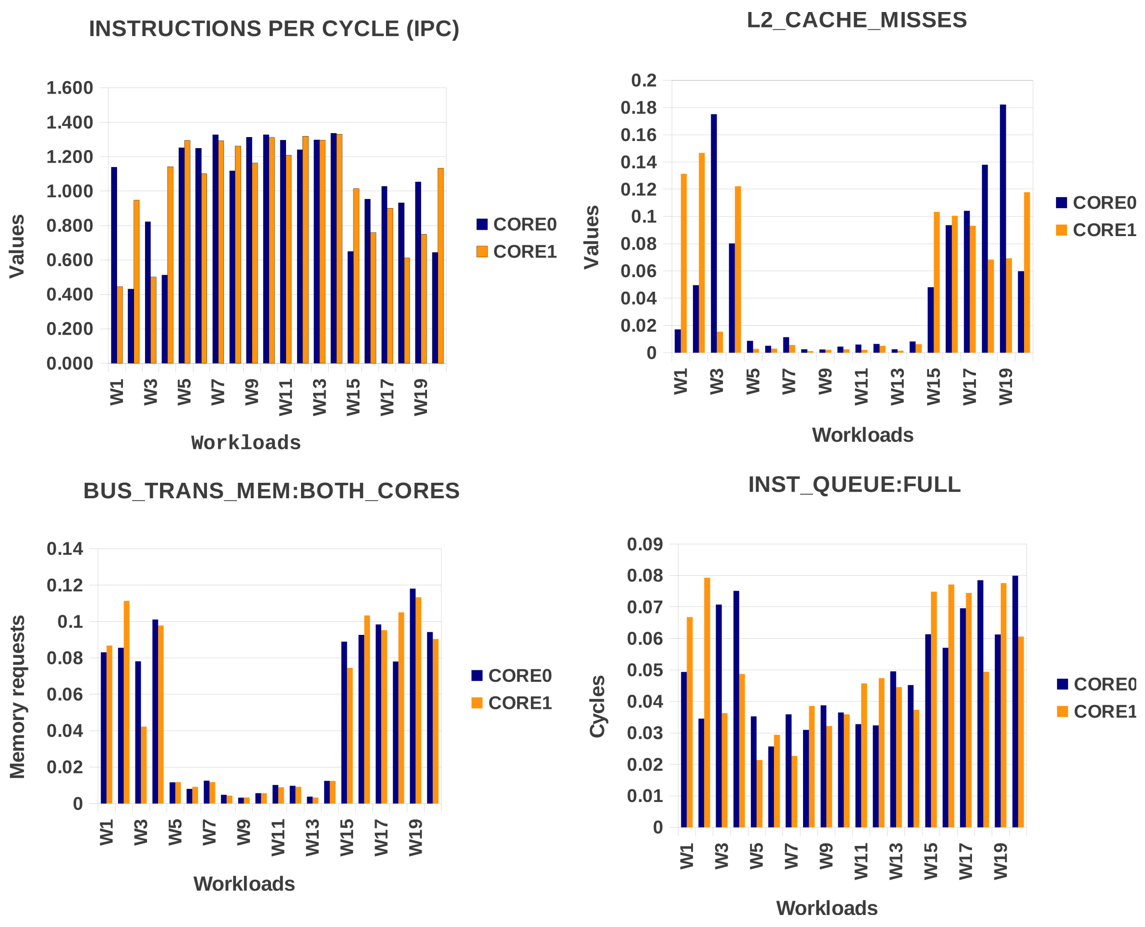

Fig. 3. Some metrics used for the CINT Speccpu2000 benchmarks characterization

First, this system call obtains a cpumask bitmap called `cpus_allowed` from the thread's task descriptor (`task_struct` structure) [6]. On the `cpus_allowed` bitmap one bit is set for each online core on which the thread can run (inherited from the process of which it is a part). Then, a bitwise AND operation is performed between the `cpus_allowed` and an input mask (`in_mask`) bitmaps to obtain `new_mask`[7].

If the core to which the current task is bound is not part of the new mask, `sched_setaffinity()` performs all the migration process[8]. The `sched_ setaffinity()` function receives a task id and an input mask as parameters. As a task id is assigned, it is necessary to obtain the corresponding task descriptor (since the `cpus_allowed` bitmask is there).

This is done through the `find_process_by_pid()` function called by `sched_ setaffinity()`. The various stages involved in the implementation of our algorithm that carries out the migration of threads between

[6]`sched_setaffinity()` calls the `cpuset_cpus_allowed()` function, which ends up calling the `task_cs()` function with a pointer to `task_descriptor` as parameter to retrieve the `cpuset` structure for the task. The `cpuset` structure holds the `cpus_allowed` cpumask bitmap. Since version 4.0.9, this structure includes the `effective_cpus` cpumask bitmap which is used instead of `cpus_allowed` for this very purpose.

[7]Through `cpumask_and()` which is called by `sched_ setaffinity()`.

[8]`sched_setaffinity()` calls the `set_cpus_allowed_ptr()` function to perform the entire process of migrating the thread when the core it is executing on is removed from the allowed bitmask.

Algorithm 1: mctopology algorithm

```
Input: Input mask in hex
Output: Length
Arguments: *buffer: user space data string
           length: length of the entered data string
Cpumask-type array: A-domain-core, B-domain-core;
procedure MCTOPOLOGY(mask);
Charater array: in_mask;
Integer variables: new_mask, num_cpus, cpui, A-domain;
Pointer to character: *mask_string;
num_cpus ⟵ get the number of cpus in the system (NR_CPU) ;   ▷ NR_CPU is a kernel variable
in_mask ⟵ get mask from user space (length of buffer);
mask_string ⟵ get address of (in_mask);
change what's in mask_string to hex;
save mask_string in new_mask;
for cpui ⟵ 0, num_cpus do
    if cpui is online then   ▷ its corresponding bit is set in the default kernel cpumask
        right shift new_mask i positions;
        do a bitwise AND operation between the;
        right shifted new_mask and a 0×1 mask;
        assign the result to A-domain;
        if A-domain=1 then
            save cpui in A-domain-core;
        else
            save cpui in B-domain-core;
        Return length
```

the two physical cores of our example CMP processor (Table 1), which we have called `sched_setmigration_newmask()`, are described next. Likewise, our algorithm can be easily extended to a system with a larger number of cores. `sched_setmigration_newmask()` is invoked within our proactive load-balancing algorithm to migrate threads when needed. These stages are:

Stage 1: The Linux kernel's `sched_setaffinity()` function (file `/kernel/sched.c`) is modified so that it accepts a task descriptor as a parameter instead of the id of the corresponding task (hence, the bulky-code `find_process_by_pid()` function is removed). Since the `cpus_allowed` bitmask can be obtained from the task descriptor, this improves code complexity (ergo, power consumption also improves [4, 38]).

The `cpus_allowed` and input mask are used by the kernel to obtain the `new_mask` mask that the scheduler checks repeatedly to know which cores are offline and perform the thread migration process accordingly.

We have called the improved function `__sched_setaffinity()` (same system-call name prefixed with double underscore[9]), which is merged into our migration `sched_setmigration_newmask()` algorithm described in detail in Section 6. As we mentioned earlier in the Introduction, our work is based on the vanilla Linux kernel version 3.1.2. From version 5.15.67, the core part of the code within `sched_setaffinity()`[10] (now found in

[9]So it is treated as an internal function and no checks are made on the user address space (the function is then executed faster) [42].

[10]The essential code for obtaining the `cpus_allowed` cpumask bitmap and migrating.

Algorithm 2: sched_setmigration_newmask() algorithm

```
   Input: current task's pid, core number
   Output: retval;                                    ▷ 0 if success, error code if failure
1  Cpumask-type array: A-domain-core, B-domain-core;
2  procedure sched_setmigration_newmaskpid, core;
3  Integer variable: retval;
4  Pointer to current task: *curr;
5  curr ⟵ get current task(pid);                      ▷ kernel's own function
6  if core=0 then
7      retval⟵call_sched_setaffinity(curr, A-domain-core);    ▷ core1 ∉ A-domain-core
8  else
9      if core=1 then
10         retval ⟵ call_sched_setaffinity(curr, B-domain-core);  ▷ core0 ∉ B-domain-core
11     else
12         Return error code
13 Return retval
```

`/kernel/sched/core.c`) is wrapped in a function that also bears the same name prefixed with double underscore (`__sched_setaffinity()`). However, `sched_setaffinity()` still uses the `find_process_by_pid()` function to obtain the thread's task descriptor.

Stage 2: Algorithm 1 shows our algorithm called mctopology (i.e., multi-core topology) which is designed to construct from an input mask in hexadecimal notation two domain masks of cores:

$\boxed{\texttt{A-domain-core}\ \text{and}\ \texttt{B-domain-core}}$

The input mask is devised to allow grouping cores that are physically adjacent (and so sharing resources, e.g., a common cache) into domain masks (i.e., scheduling domains) such that threads should preferably be moved between the cores in a domain. To pass the input mask to the kernel to be processed, the `/proc/topology` directory and the `user_cpumask_input` virtual file are created as a means of communication between the user space and the kernel space.

In lines 10 and 11, the number of cores in the system (taken straight from the kernel variable `NR_CPU`) is stored in the `num_cpus` variable and the input mask is stored in the `in_mask` variable respectively. In lines 12 − 14, `in_mask`'s address is assigned to the `mask_string` pointer and the string value stored at the address where `mask_string` is pointing is changed into a hexadecimal format and stored in the `new_mask` variable. Lines 15 − 28 comprise the method for grouping cores into scheduling domains so that a thread is migrated to a core within the same domain. In this way, the thread's information stored in the cache that is shared among the cores within the domain is not moved to another system cache (resulting in lower migration cost).

This section of code first tests if the bit that corresponds to the core at position i (cpui) is set in the default kernel `cpu_allowed` cpumask (i.e., if the core is online). If so, a right shift operation on `new_mask` of i positions is performed and a bitwise AND operation between the right-shifted `new_mask` and a `0x1` mask (to select adjacent cores) is executed. Then, the result is assigned to the `A-domain` variable (similarly, a `0x2` mask could have been used for the `B-domain` variable).

Depending on the result that has been stored in `A-domain`, the core is saved in either the mask `A-domain-core` or `B-domain-core`. Since our system has only two (online) cores, the domain masks `A-domain-core` and `B-domain-core` store corei (i=0) and corei (i=1) respectively. Depending on the number of cores in a multi-core system, more complex layouts of scheduling domains can be obtained.

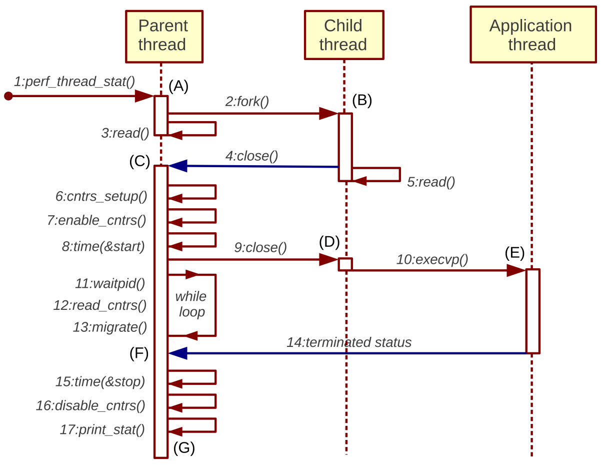

Fig. 4. Sequential diagram for our PLB algorithm

As the number of cores increases, the setting of an input mask so that the selection of the adjacent cores be optimal is a matter of key importance.

Stage 3: An algorithm to update the current task's cpumask bitmap with either the mask `A-domain-core` or `B-domain-core` is devised and implemented as a system call. It is invoked within our Proactive Load-Balancing algorithm to migrate threads when needed. We named it:

```
sched_setmigration_newmask()
```

Our `sched_setmigration_newmask()` algorithm receives, as a parameter, the number of the core to which the task must be migrated. Depending on the core number, the algorithm chooses either the `A-domain-core` or `B-domain-core` domain mask. The algorithm then calls the `__sched_setaffinity()` function with the current task descriptor and the new mask as parameters, so the current task is migrated to the target core.

For our example dual-core system (which has two physical cores and hyperthreading disabled), the default kernel cpumask is 0x11 (both cores are online), so when the `A-domain-core` domain mask is chosen within our `sched_setmigration_newmask()` algorithm, the current task is forced to leave core0 and migrate to core1. Similarly, when the `B-domain-core` domain mask is chosen, the current task is forced to leave core1 and migrate to core0. This algorithm is shown in Algorithm 2.

In line 7, the current task descriptor is obtained first. In lines 8 − 17, a domain mask for the `__sched_setaffinity()` function is chosen as its second parameter according to the core number that is passed to `sched_setmigration_newmask()`. Since our system has only two (online) cores, there is one core per domain.

On Linux, threads are always migrated from one run queue to another. Before being migrated, the kernel must suspend the execution of the thread running on the local core and save its task context (including performance statistics) accordingly. These actions are part of a context switching procedure performed in kernel mode. For a thread, its task context is obtained from its task descriptor (`task_struct` structure) and CPU registers[11].

The task descriptor contains all of the data needed to keep track of the thread, whether it is running or not. Some of the primary fields it includes are[12]: A `thread_info` structure that holds all requiered processor-specific low-level information about the thread, a `__state` variable, a `*stack` pointer to its kernel-mode stack, a `*mm` pointer to its virtual address space (`mm_struct` structure, also called memory descriptor), and a `thread_struct` structure that holds the architecture-specific state of the thread.

The thread's virtual address space[13] is divided into several regions of type `vm_area_struct` each of which contains different information of the running thread such as its user-mode stack, code, data, and so on. When entering kernel mode to run the context switcher, the instruction pointer (EIP), the status register (EFLAGS, also known as a condition-code register or CCR), the user stack pointer (EBP), the segment selector of the user data segment (__USER_DS) and the segment selector of the user code segment (__USER_CS) of the thread being migrated are saved automatically

[11]It is not necessary to save the full state of the machine for a thread, as it is using the same memory, program code, files and devices as the process of which it is a part. A thread must maintain only some state information of its own.

[12]From the latest stable kernel version 5.19.5.

[13]The process that spawns the thread initially shares its address space with it. Later, when the thread modifies or writes to a part of this space, a copy of that part is made for the thread itself (known as the Copy-On-Write technique) [34].

Algorithm 3: proactive load-balancing algorithm Part 1

```
Input: application's executable code and its own parameters
Output: performance events' statistics of the monitored application thread
procedure perf_thread_stat() ;                ▷ input is entered at the command line
child_pid ⟵ fork()
if child_pid $< 0$ then                               ▷ the parent does this
    failed to fork
if child_pid $\neq 0$ then                            ▷ the child does this
    do ⟶ preparatory steps prior to launching the application thread;
    close one end of the pipe 1;
    read the open end of pipe 2;
    execvp ⟵ application's name;
read the open end of pipe 1;                          ▷ the parent does this
do ⟶ the counters' settings;                                ▷ See Table 2
ioctl ⟵ RESET, $fd_i$;              ▷ a file descriptor (fd) selects a counter
ioctl ⟵ ENABLE, $fd_i$;
$t_0$ ⟵ time(& start);                                  ▷ the start runtime
close one end of pipe 2;
$k \longleftarrow 0$;
waitpid ⟵ child_pid, WNOHANG ; ▷ in order to not suspend the execution of the parent thread
```

on the kernel-mode stack[14]. The context switcher then saves in the `thread_struct` structure housed in the task descriptor, the remainder of CPU registers that hold the state of the machine at the time the core is deallocated from the thread.

Thus, since the thread's task descriptor is moved from the local run queue to the target run queue, the thread's task context can be restored and its execution resumed (a thread is always scheduled from the run queue on to the core in user mode [18]).

CPU cycles consumed in all this moving that the kernel does to migrate threads result in pure overhead, because no useful work (IPC) is done [34, 8, 22, 23].

In this light, our PLB algorithm maintains two different threads from each workload running concurrently as long as it results in the least shared resource contention, and therefore, to the same extent, thread migration is reduced; ultimately preventing system performance from declining.

[14]A context switch is always initiated by an interrupt. The interrupt mechanism saves automatically this data on the kernel-mode stack [18, 3, 42, 31].

In the next section, we first illustrate sequentially how our algorithm works, and then explain its pseudocode line by line.

6 Proactive Load-Balancing (PLB) Algorithm

6.1 Sequential Diagram

A sequential diagram to depict the interaction of the various threads involved in our algorithm is shown in Fig. 4. In this diagram the leading function (parent thread) called `perf_thread_stat()` appears in the upper left and tagged with the number 1. When the algorithm starts, it calls the function `perf_thread_stat()` which spawns a child thread using the kernel's standard `fork()` function.

Next, the child thread starts executing in parallel to the parent until the parent thread stops and waits for the child to complete the preparatory steps prior to launching the application (which will be detailed in the next section) before carrying out the counters' setup. Using an inter-thread communication technique known as pipes, the parent and child get synchronized by sending

Algorithm 4: proactive load-balancing algorithm Part 2

```
1  while do
2      if $k = 0$ then                                         ▷ first pass inside the while loop
3          refval1⟵ ipcval1;
4          refval2⟵ ipcval2
5      if $k \neq 0$ then
6          if refval1 $<$ ipcval1 then
7              refval1 ⟵ ipcval1
8          if refval2 $<$ ipcval2 then
9              refval2 ⟵ ipcval2
10     if count2 $\neq$ 0 && ipcval1 $<$ refval1 && mf1 $\neq$ 1 then
11         res ⟵ sched_setmigration_newmask(child_pid, 1);  ▷ our algorithm to migrate threads
12         mf1 ⟵ 1;
13         mf2 ⟵ 0;
14         refval1 ⟵ 0;                                     ▷ refval1 is reset
15     if count3 $\neq$ 0 && ipcval2 $<$ refval2 && mf2 $\neq$ 1 then
16         res⟵ sched_setmigration_newmask(child_pid, 0);   ▷ our algorithm to migrate threads
17         mf2⟵ 1;
18         mf1⟵ 0;
19         refval2⟵ 0 ;                                     ▷ refval2 is reset
20     $k \longleftarrow k + 1$
21 $t_1$ ⟵ time(&stop);                                     ▷ the end runtime
22 ioctl⟵ DISABLE, $fd_i$;
23 read the final values of the statistics;
24 print the performance events' statistics of the monitored application thread;
```

messages between them. A null message is sent from the end of a pipe that gets closed, to the end that remains open to read the message. The `close()` function is used to close one end of a pipe, whereas the `read()` function is used to read at the other end.

This is shown through the points A-C. When the parent thread receives the null message from the child, the parent resumes execution and performs the counters' setup. Straight afterwards, the parent thread first resets and enables the counters and then starts measuring the runtime parameter through the `time()` function.

Meanwhile, the child thread waits for the parent to finish these steps. Right after the parent thread has started measuring the runtime parameter, it sends a null message to the child by closing one end of a second pipe (D).

The child thread reads the message at the other end of this pipe, resumes execution, and launches the application by means of the `execvp()` function. Now, this is shown through the points C-E. While the application is running, the parent thread reads the counters and stores their values into data arrays within a `while` loop.

The `while` condition includes a `waitpid()` function set to wait for the application to terminate without suspending the parent thread execution (by using the `WNOHANG` option). Thus, the parent thread keeps gathering statistical information from the counters while the application is running.

Based on these statistics, the parent thread decides whether it is convenient to migrate the current application thread. When the application ends, the terminated status for the child specified by its pid is immediatly available.

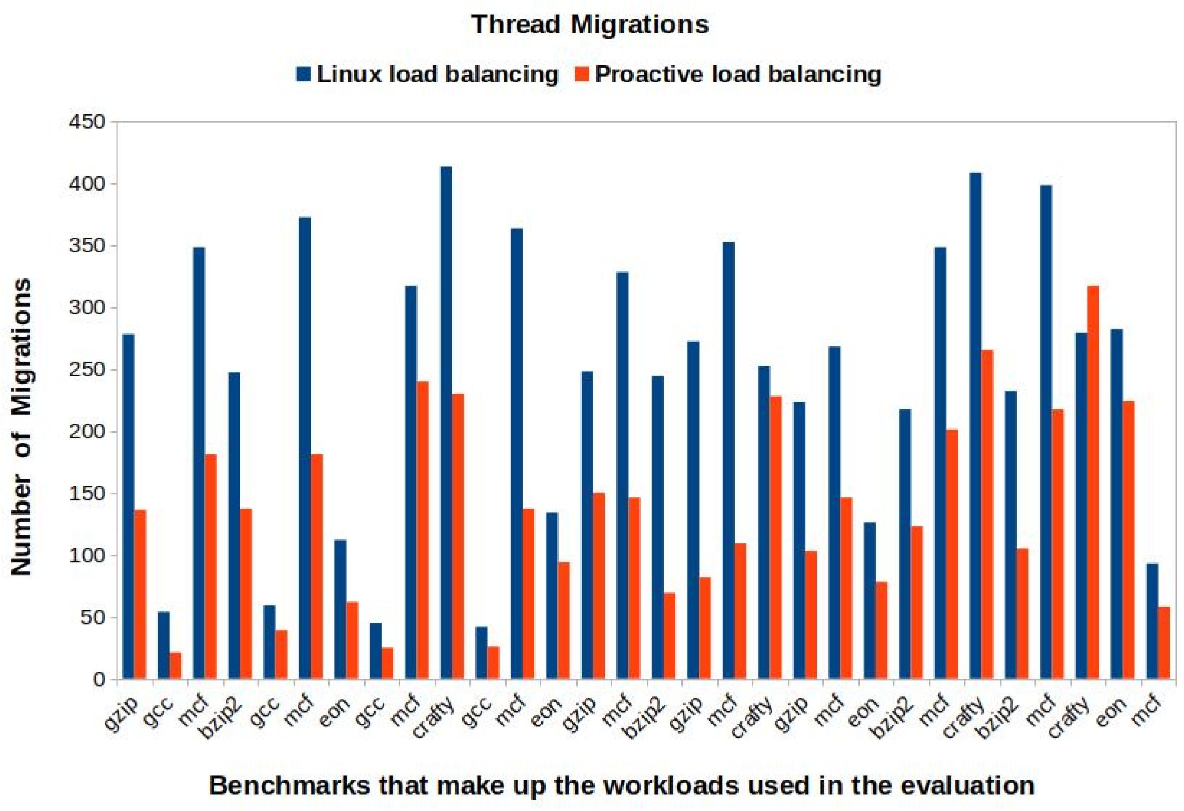

Fig. 5. Number of thread migrations obtained for both load balancers after 10 runs of each workload

Then, the parent thread leaves the `while` loop, stops measuring the runtime, and disables the counters. The `ioctl()` function was used to reset, enable, and disable the counters.

Finally, the parent thread saves the collected statistics in dedicated files, prints them on the screen, and exits. See points E-G. Next, we present our proactive load-balancing algorithm split into two parts: Algorithm 3 and Algorithm 4.

6.2 Detailed Description

Our PLB algorithm requires a tight synchronization between the parent, child and application threads as exposed so far. Algorithm 3 details the synchronization between these threads as well as the first part of the `while` loop depicted above.

In lines 4 – 8, the parent thread spawns a child thread through the `fork()` function and checks if the returned `pid` has a valid value.

At this point a branch occurs: in line 9, the child begins executing the preparatory steps prior to launching the application thread, and in line 14, the parent thread waits for the child to finish.

In line 10, just after the child thread has completed these steps, it sends a null message to the parent by closing one end of pipe 1. In line 11, the child waits for the parent thread to carry out the counters' setup.

Then, in lines 15 – 18, the counters are setup, reset and enabled by the parent thread, whereupon it begins measuring the runtime parameter. In line 19, the parent sends a null message to the child by closing one end of pipe 2, so in line 12, the child launches the application by means of the `execvp()` function.

Table 4. Percentage of reduction in the number of thread migrations for each workload

Thread Migrations									
Workload	Benchmarks (threads)								
W1	**gzip**			**gcc**			**mcf**		
	278	136	**51.1%**	54	21	**61.1%**	348	181	**47.8%**
W2	**bzip2**			**gcc**			**mcf**		
	247	137	**44.5%**	59	39	**35.4%**	372	181	**51.4%**
W3	**eon**			**gcc**			**mcf**		
	112	62	**44.6%**	45	25	**44.4%**	317	240	**24.3%**
W4	**crafty**			**gcc**			**mcf**		
	413	230	**51.6%**	42	26	**38.1%**	363	137	**62.2%**
W15	**eon**			**gzip**			**mcf**		
	134	94	**29.9%**	248	150	**39.5%**	328	146	**55.5%**
W16	**bzip2**			**gzip**			**mcf**		
	244	69	**71.7%**	272	82	**69.8%**	352	109	**69.0%**
W17	**crafty**			**gzip**			**mcf**		
	252	228	**9.5%**	223	103	**53.8%**	268	146	**45.1%**
W18	**eon**			**bzip2**			**mcf**		
	126	78	**38.1%**	217	123	**43.3%**	348	201	**42.2%**
W19	**crafty**			**bzip2**			**mcf**		
	408	265	**35.0%**	232	105	**54.7%**	398	217	**45.5%**
W20	**crafty**			**eon**			**mcf**		
	279	317	**-13.6%**	282	224	**20.6%**	93	58	**37.6%**

As mentioned earlier, the child performs a couple of preparatory steps prior to launching the application thread which consist in:

1. First calling a dummy `execvp()` that always fails in order to avoid Global Offset Table (GOT) and Procedure Linking Table (PLT) entry relocation overhead on the real `execvp()` [19, 27].

   These processor-specific tables assist the dynamic linker in finding the absolute addresses for position-independent function calls, such as `execvp()`. Therefore, as all this action is performed in advance, to launch the real `execvp()` takes much fewer steps.

2. Setting the close-on-exec flag (`FD_CLOEXEC`) associated with the file descriptor representing the open end of pipe 2, so this end will be automatically (and atomically) closed when the `execvp()` succeeds (since `execvp()` does not return when successful).

On the other hand, in line 20, the control variable `k` that is used to indicate the number of times the algorithm enters into the `while` loop is initialized to 0. In line 21, the `waitpid()` function is configured using the `WNOHANG` option along with the child's pid in order not to suspend the execution of the parent thread while the application is running. In lines 22 − 26, the `while` condition is set and the counts of the events represented by their corresponding file descriptors (Table 2) are stored into dedicated array variables. In lines 27 − 38, the IPC is calculated from the instructions retired and unhalted core cycles statistics for both the core0 and core1 at runtime.

The decimal part of the IPC values thus obtained is truncated to its hundredth part; this in order to make them more meaningful. Algorithm 4 presents the second part of the `while` loop which contains the reasoning for deciding to migrate the application threads (i.e., the decision-making part).

In lines 39 − 49, for the first pass inside the `while` loop, the IPC values that have been previously calculated are assigned to IPC-threshold variables (refval1 and refval2). If it is a subsequent pass, it checks whether the IPC-threshold values are lower than the corresponding new IPC values for core0 and core1.

If so, the new IPC values are assigned to the IPC-threshold variables (i.e., runtime-updated IPC thresholds are used). On the other hand, in lines 51 − 62, if the new IPC values are lower than the IPC-threshold values, it means that a significant contention exists between the co-runner threads.

So the current IPC-threshold values are maintained. In line 51, the IPC value of the thread running on core0 (`ipcval1`) is checked, if it is lower than its corresponding IPC-threshold value (`refval1`), the thread is migrated to core1 by means of our `sched_setmigration_newmask()` algorithm.

Similarly, in line 57, the IPC value of the thread running on core1 (`ipcval2`) is checked, if it is lower than its corresponding IPC-threshold value (`refval2`), the thread is migrated to core0[15]. Therefore, as soon as a significant decrease in the thread's IPC is observed at runtime, our algorithm reacts proactively trying to keep this parameter to its previous higher value by migrating the thread and thus avoiding contention for shared resources with its co-runners.

[15]This is completely in line with our proposed thread migration model. For a system with a larger number of cores, our model would simply have more choices of cores to migrate to.

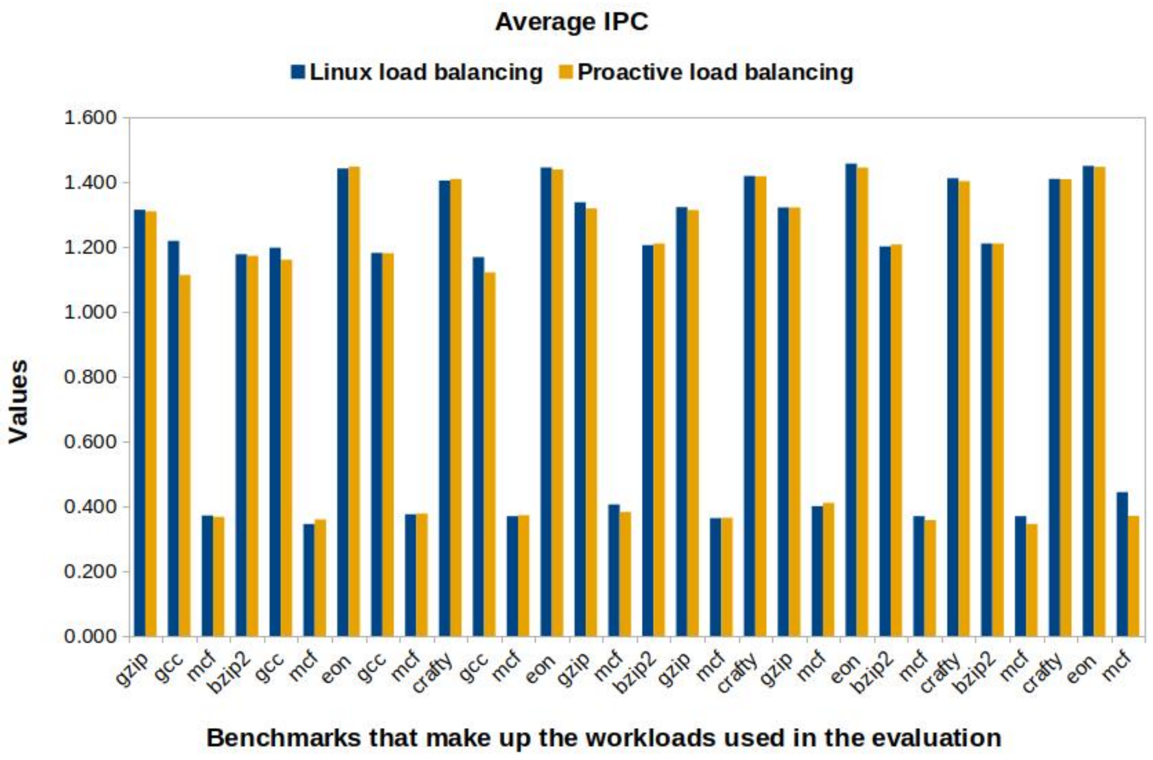

Fig. 6. Average IPC obtained for both load balancers after 10 runs of each workload

As previously stated, the optimal couples of co-runner threads are searched in this way at runtime. The `mf1` and `mf2` variables are used to ensure that a thread cannot be migrated to the same core where it is currently running. In line 63, the control variable `k` is incremented by 1 for each pass through the `while` loop.

Finally, when the application terminates, the `while` condition is no longer met, so in lines 65 – 68, the parent thread stops measuring the runtime parameter, disables the counters, reads the final values of the statistics, and prints these final values along with those previously obtained during the execution of the application.

7 Evaluation Experiments

This section details the steps performed to evaluate our proactive load-balancing algorithm: First, workload implementations of CINT speccpu 2000 benchmarks [36] which simulate application threads are carried out. Three-benchmark workloads that have adequate stress capacity for the example multi-core system used (Table 1) are employed. Second, perf_event is used to configure the PMU built into each processor core in order to collect the instructions retired and unhalted core cycles hardware events from which we calculate the Instructions Per Cycle (IPC) statistic.

As explained in earlier sections, IPC is the primary metric we employed to manage the migration of process threads. Although there are several other events collectable such as L2 cache misses, rejected L2 cache requests (by the bus queue), completed memory transactions on the system bus, and so on, IPC is a good performance metric to keep simplicity in our conception. The PMU is also configured to collect the cpu-migrations software event at the same time (as described in Section 3).

Third, as mentioned in Section 2, we developed a workload-launcher tool in order to synchronously launch the implemented workloads on the CMP system. The workload-launcher picks the different bechmarks that make up a workload and launches them simultaneously to execution. As soon as a workload finishes, the benchmarks that compose the next workload are launched. When the last workload is executed, the workload-launcher starts over by executing each workload again in a second round. It stops after 10 rounds are completed (i.e., each workload is run 10 times). As discussed earlier, 10 workloads were selected from the 20 that we had previously characterized to stress the CMP system used.

Fourth, in the manner described in the previous step, each of the selected workloads is run 10 times on Linux without modifying the scheduler, and the number of instructions retired, unhalted core cycles and thread migrations are counted at runtime. Fifth, our PLB algorithm is then merged into the Linux scheduler and each of these workloads is run 10 times again. The same events are counted at runtime. Finally, the statistics values obtained are plotted on a bar graph and analyzed. This bar graph is shown in Fig.5.

8 Results

The number of thread migrations obtained after running each workload 10 times on the example CMP system with the Linux OS installed, first with the vanilla (standar) kernel and then with the PLB algorithm merged into the Linux kernel scheduler is plotted in the bar graph in Fig. 5, which also shows the benchmarks that make up each workload. From this figure it can be seen that the number of migrations performed is significantly reduced when using proactive load balancing.

Table 4 shows in detail the percentage by which the number of migrations decreased for each constituent benchmark (mimicking a software thread) of the workloads used. In this table, below each benchmark, there are three small boxes. In the first box, we have the resulting number of thread migrations for the vanilla Linux scheduler, in the second, the corresponding number for our proactive load-balancing algorithm, and in the third, the percentage by which the migrations decreased. Our PLB algorithm reduces the number of thread migrations by up to 71.7% (`bzip2` in W16). There is only one case (`crafty` in W20) where there is a 13.6% increase. For this particular combination of benchmarks that make up W20, both `eon` and `mcf` contend, one at a time, against `crafty` for shared resources with great intensity.

Therefore, our algorithm is forced to migrate `crafty` quite often. For these workloads, we also compared the average IPC obtained for both the vanilla and modified scheduler instances. The resulting numbers for proactive load balancing are practically the same as those of the vanilla instance. This is exposed in the bar graph in Fig.6. From Table 4, we obtain the total number of migrations for both the vanilla scheduler (M_T) and proactive load balancing (m_T): Vanilla scheduler:

$$M_T = \sum_{i=1}^{30} M_i = 7354. \tag{1}$$

Proactive load balancing:

$$m_T = \sum_{i=1}^{30} m_i = 4130. \tag{2}$$

Thus, the total reduction in the number of migrations (T_r) is:

$$T_r = M_T - m_T = 7354 - 4130 = 3224. \tag{3}$$

That is, in total there were 3224 fewer migrations.

Therefore, on average the percentage by which the number of migrations was reduced (A_r) is:

$$A_r = \left(\frac{3224}{7354} \times 100\right)\% = 43.8\%. \tag{4}$$

Without degradation of performance.

9 Conclusion and Future Work

We have presented our proactive load-balancing (PLB) algorithm designed to avoid performance drop in CMP systems due both to an excessive number of thread migrations and shared resource contention. Our PLB algorithm proactively decides whether a running thread must be migrated from its current core to another active core based on performance data read at runtime from system counters (PMU) that we configured to collect counts of selected events.

While the software threads are running concurrently (co-running threads) on a CMP processor, the Instructions Retired and Elapsed Cycles events are collected and used to obtain the IPC parameter which shows the system performance. In the same way, the cpu-migrations event is also gathered to know the number of migrations performed on each thread.

According to the literature, shared resource contention by co-running threads is the most important cause of performance drop. Hence, our PLB algorithm is primarily designed to proactively avoid resource contention. On the example CMP machine, our algorithm maintains two different threads from each workload running concurrently as long as it results in the least shared resource contention, and therefore, to the same extent, thread migration is reduced.

Our PLB algorithm avails itself of the Linux kernel's perf_event subsystem to configure each core's PMU, read event counts and monitor software threads. The perf_event subsystem represents an easy access to hardware counters to Linux, which are a key resource for improving system performance. Unfortunately, there is a lack of literature and limited online documentation available for this monitoring tool.

Therefore, an important aspect of our research work is that we have excelled at dredging up most of the vague and obscure configuration facts, and thus shedding light on how to use and set up perf-event; namely, how to set it up in order to count different events simultaneously at runtime. We have introduced a comprehensive view of the methodology used for conducting research to adapt an operating system such as Linux to the modern multi-core architectures, which is a theme of great relevance and interest among computer scientists today. Overall, our results have shown that the number of migrations performed on the threads (benchmarks) that make up the workloads used is significantly reduced (by 43.8% on average) without harming system performance when our proposed PLB algorithm is utilized. To design A.I. algorithms that can be merged into Linux to perform smart scheduling of co-running threads in multi-core architectures so that system performance improves is an interesting avenue for future work.

Acknowledgments

We thank Instituto Politécnico Nacional (IPN) for funding this work through the projects:

– SIP20231335: Open Technologies for the Development of SoCs based on the Lagarto-I RISC-V Processor and Linux OS for Academia and Research.

– SIP20231075: Lagarto SBC: Computer on module based on the Lagarto processor.

Ulises Revilla-Duarte thanks Consejo Nacional de Ciencia y Tecnología de México (CONACYT) for the Ph.D. Scholarship granted and IPN for the facilities provided to carry out this research work.

References

1. **Arpaci-Dusseau, R. H., Arpaci-Dusseau, A. C. (2018).** Operating systems: Three easy pieces. Chapter 10: Multiprocessor Scheduling, CreateSpace Independ-ent Publishing Platform, pp. 103–112.
2. **Blagodurov, S., Fedorova, A. (2011).** User-level scheduling on NUMA multicore systems under linux. Proceedings of the Linux Symposium, pp. 81–92.
3. **Bovet, D. P., Cesati, M. (2005).** Understanding the Linux kernel. O'Reilly.

4. **Brandolese, C., Fornaciari, W., Salice, F., Sciuto, D. (2002).** The impact of source code transformations on software power and energy consumption. Journal of Circuits, Systems and Computers, Vol. 11, No. 5, pp. 477–502. DOI: 10.1142/s0218126602000586.

5. **Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., Namyst, R. (2010).** hwloc: A generic framework for managing hardware affinities in HPC applications. Proceedings of the 18th Euromicro International Conference on Paralel, Distributed and Network-based Processing, pp. 180–186. DOI: 10.1109/pdp.2010.67.

6. **Chandra, D., Guo, F., Kim, S., Solhin, Y. (2005).** Predicting inter-thread cache contention on a chip multi-processor architecture. Proceedings of the 11th International Symposium on High-Performance Computer Architecture, pp. 340–351. DOI: 10.1109/HPCA.2005.27.

7. **Clarkdale (2007).** Clarkdale microprocessor.

8. **Constantinou, T., Sazeides, Y., Michaud, P., Fetis, D., Seznec, A. (2005).** Performance implications of single thread migration on a chip multi-core. ACM SIGARCH Computer Architecture News, Vol. 33, No. 4, pp. 80–91. DOI: 10.1145/1105734.1105745.

9. **Corporation, N. D. (2019).** Tomoyo linux. tomoyo.sourceforge.net.

10. **Eranian, S., Gourion, E., Moseley, T., Bruijn, W. (2015).** Linux kernel profiling with perf. perf.wiki.kernel.org/index.php/Tutorial.

11. **Garcia-Garcia, A., Saez, J. C., Prieto-Matias, M. (2018).** Contention-aware fair scheduling for asymmetric single-isa multicore systems. IEEE Transactions on Computers, Vol. 67, No. 12, pp. 1703–1719. DOI: 10.1109/tc.2018.2836418.

12. **Goglin, B. (2017).** On the overhead of topology discovery for locality-aware scheduling in HPC. Proceedings of the 25th Euromicro International Conference on Paralel, Distributed and Network-based Processing, pp. 186–190. DOI: 10.1109/pdp.2017.35.

13. **Gouicem, R. (2020).** Thread scheduling in multi-core operating systems: How to understand, improve and fix your scheduler. Ph.D. thesis, Sorbonne Université, France.

14. **Harris, J. A., Cordani, J. (2002).** Schaum's outline of operating systems. Chapter 2: Process Management, McGraw-Hill, pp. 14–23.

15. **Herdrich, A., Illikkal, R., Iyer, R., Newell, D., Chadha, V., Moses, J. (2009).** Rate-based QoS techniques for cache/memory in CMP platforms. Proceedings of the 23th International Conference on Supercomputing, pp. 479–488. DOI: 10.1145/1542275.1542342.

16. **Intel (2007).** Intel®core™2 duo processor e6550. https://ark.intel.com/content/www/us/en/ark/products/30783/intel-core-2-duo-processor-e6550-4m-cache-2-33-ghz-1333-mhz-fsb.html.

17. **Intel (2011).** Intel®64 and IA-32 architectures software developer manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

18. **John, O. (2001).** Operating systems with linux. Chapter 3: Process Manager, Palgrave, pp. 39–67.

19. **Jones, M. (2008).** Anatomy of linux dynamic libraries. developer.ibm.com/tutorials/l-dynamic-libraries/.

20. **Jung, J., Shin, J., Hong, J., Lee, J., Kuo, T. W. (2017).** A fair scheduling algorithm for multiprocessor systems using a task satisfaction index. Proceedings of the International Conference on Research in Adaptive and Convergent Systems, pp. 269–274. DOI: 10.1145/3129676.3129736.

21. **Lim, G., Min, C. W., Eom, I. Y. (2012).** Load-balancing for improving user responsiveness on multicore

embedded systems. Proceedings of the Linux Symposium, pp. 25–34. DOI: 10.48550/arXiv.2101.09359.

22. **Lozi, J. P., Lepers, B., Funston, J., Gaud, F., Fedorova, A., Quéma, V. (2016).** The linux scheduler: A decade of wasted cores. Proceedings of the Eleventh European Conference on Computer Systems (EuroSys'16), pp. 1–16. DOI: 10.1145/2901318.2901326.

23. **Lozi, J. P., Lepers, B., Funston, J., Gaud, F., Fedorova, A., Quéma, V. (2016).** Your cores are slacking off–or why os scheduling is a hard problem. USENIX Association, Vol. 41, No. 4, pp. 6–13.

24. **Marinakis, T., Haritatos, A. H., Nikas, K., Goumas, G. I., Anagnostopoulos, I. (2017).** An efficient and fair scheduling policy for multiprocessor platforms. pp. 1–4. DOI: 10.1 109/ISCAS.2017.8050758.

25. **Mosberger, D., Eranian, S. (2002).** IA-64 linux kernel: Design and implementation. Prentice Hall.

26. **Oprofile (2013).** Oprofile - a system profiler for linux. oprofile.sourceforge.io.

27. **Oracle (2008).** Linkers and library guide. docs .oracle.com/cd/E23824_01/html/819-0690/toc. html.

28. **Pathania, A., Venkataramani, V., Shafique, M., Mitra, T., Henkel, J. (2016).** Distributed fair scheduling for many-cores. pp. 379–384.

29. **Permon2 (2013).** perfmon2 - improving performance monitoring on linux. perfmon2.sourceforge.net.

30. **Sáez, J. C., Gómez, J. I., Prieto, M. (2008).** Improving priority enforcement via non-work-conserving scheduling. Proceedings of the 37th International Conference on Parallel Processing, pp. 99–106. DOI: 10.110 9/ICPP.2008.38.

31. **Salzberg-Rodriguez, C., Fischer, G., Smolski, S. (2005).** The Linux®kernel primer: A top-down approach for x86 and powerpc architectures. Chapter 3: Processes: The Principal Model of Execution, Prentice Hall, pp. 77–178.

32. **Shi, Q., Chen, T., Hu, W., Huang, C. (2009).** Load balance scheduling algorithm for CMP architecture. Proceedings of the International Conference on Electronic Computer Technology, pp. 396–400. DOI: 10.1109/icect.2009.74.

33. **Siddha, S., Pallipadi, V., Mallick, A. (2005).** Chip multiprocessing aware linux kernel scheduler. Proceedings of the Linux Symposium, pp. 193–204.

34. **Silberschatz, A., Baer-Galvin, P., Gagne, G. (2007).** Operating system concepts with java. John Wiley and Sons.

35. **Silberschatz, A., Baer-Galvin, P., Gagne, G. (2018).** Operating system concepts. Chapter 5: CPU Scheduling, John Wiley and Sons, pp. 220–227.

36. **SPEC (2007).** Standard performance evaluation corporation. www.spec.org.

37. **Tanenbaum, A. S., Herbert, B. (2015).** Modern operating systems. Chapter 8: Multiple Processor Systems, Person Education, pp. 520–539.

38. **Tiwari, V., Malik, S., Wolfe, A., Lee, M. T. C. (1996).** Instruction level power analysis and optimization of software. Proceedings of 9th International Conference on VLSI Design, pp. 1–18. DOI: 10.1109/icvd.1996.489624.

39. **Vogl, S., Eckert, C. (2012).** Using hardware performance events for instruction-level monitoring on the x86. Proceedings of EuroSec'12, 5th European Workshop on System Security.

40. **Weaver, V. (2013).** Linux perf_event features and overhead. FastPath: Second International Workshop on Performance Analisys of Workload Optimized Systems, pp. 1–6.

41. **Weaver, V. (2013).** perf_event – programming guide. web.eece.maine.edu/~vweaver/project s/perf_events/programming.html.

42. **Wolfgang, M. (2008).** Professional Linux®kernel architecture. Wiley Publishing.

43. **Zhang, X., Zhong, R., Dwarkadas, S., Shen, K. (2012).** A flexible framework for throttling-enabled multicore managment. Proceedings of the 41st International Conference on Parallel Processing, pp. 389–398.

44. **Zhuravlev, S., Blagodurov, S., Fedorova, A. (2010).** Addressing shared resource contention in multicore processors via scheduling. ACM SIGARCH Computer Architecture News, Vol. 38, No. 1, pp. 129–142. DOI: 10.1145/1735970.1736036.

Article received on 25/11/2022; accepted on 12/06/2024.
** Corresponding author is Marco A. Ramírez-Salinas.*